cpp-validator (C++ Data Validation Library)

Motivation

A common issue when receiving data either by application from user interface or by server remotely from clients is verifying data integrity and/or specific constraints. A typical approach for implementing complex data validation is writing nested *if-conditions* or chained invocations of partial validation methods. Declarations of the constraints could become intermixed with their implementations. Thus, validation would spread over the code which makes it hard to maintain it. Requirement to construct and show error messages could make everything still more complicated.

cpp-validator library allows one to declare data constraints with clean statements in certain points of code and apply them in other points of code on demand. If needed, the validation error messages are automatically constructed by taking into account the user's locale.

Minimal example:

```
// define validator
auto v1=validator(
      _[key1][key1_1][key1_1_1](gt,100),
      _[key2][key2_1](value(ne,"UNKNOWN") ^AND^ size(lte,32))
  );

// validate objects
Class1 obj1;
Class2 obj2;

validate(obj1,v1);
validate(obj2,v1);
```

Introduction

cpp-validator is a modern C++ header-only library for validation of variables, objects and containers.

The library can be used to validate: - plain variables; - properties of objects, where a property can be accessed either as object's variable or object's method; - contents and properties of containers; - nested containers and objects; - heterogeneous containers such as pairs and tuples; - trees; - results of evaluations or value transformations.

Basic usage of the library includes two steps:

- first, define a validator using almost declarative syntax;
- then, apply the validator to data that must be validated and check the results.

The library is suitable for both *post-validation* and *pre-validation*. *Post-validation* stands for validating the object that is already populated with the data. *Pre-validation* stands for validating the data before updating the object. The same validator declaration can be used in both cases.

There are a lot of options for cpp-validator extension and customization. During validation a text report describing an error can be constructed. Reports can be widely customised and translated to desired languages.

The library is tested with *Clang*, *GCC* and *MSVC* compilers that support *C++14* or *C++17* standards. Tested operating systems include *Linux*, *Windows*, *macOS*, *iOS* and *Android* platforms.

For more details see *Documentation*.

Some examples of basic usage

Check nested container elements and print report

```
// define compound validator of nested container elements
auto v=validator(
            _["field1"][0](lt,100),
            _["field1"][1](in,range({10,20,30,40,50})),
            _["field2"](exists,false),
            _["field3"](empty(flag,true))
        );

error_report err;

// apply validator to container and construct error message

std::map<std::string,std::map<size_t,size_t>> nested_map={
        {"field1",{{1,5},{2,50}}},
        {"field3",{}}
    };
validate(nested_map,v,err);
if (err)
{
   std::cerr << err.message() << std::endl;
   /* prints:

   "element #1 of field1 must be in range [10, 20, 30, 40, 50]"

   */
}
```

Check if value is greater than constant

Error as an argument.

```
// define validator
auto v=validator(gt,100);

// validate variables
error err;
```

```
validate(90,v,err);
if (err)
{
  // validation failed
}

validate(200,v,err))
if (!err)
{
  // validation succeeded
}
```

Error as an exception.

```
// define validator
auto v=validator(gt,100);

// validate variables

try
{
    validate(200,v); // succeed
    validate(90,v); // throw
}
catch (const validation_error& err)
{
    std::cerr << err.what() << std::endl;
    /* prints:

    "must be greater than 100"

    */
}
```

Explicit applying of a validator.

```
// define validator
auto v=validator(gt,100);

// apply validator to variables

int value1=90;
if (!v.apply(value1))
{
  // validation failed
}

int value2=200;
```

```
if (v.apply(value2))
{
  // validation succeeded
}
```

Check if string is greater than or equal to other string and size of the string is less than constant

```
// define validator
auto v=validator(
  value(gte,"sample string"),
  size(lt,15)
);

// apply validator to variables

std::string str1="sample";
if (!v.apply(str1))
{
  // validation failed, string is less than sample string
}

std::string str2="sample string+";
if (v.apply(str2))
{
  // validation succeeded
}

std::string str3="too long sample string";
if (!v.apply(str3))
{
  // validation failed, string is too long
}
```

Check if value is in interval and print report

```
// define validator
auto v=validator(in,interval(95,100));

// apply validator to variable and construct validation error message

error_report err;

size_t val=90;
validate(val,v,err);
if (err)
{
    std::cerr << err.mesage() << std::endl;
    /* prints:
```

```
    "must be in interval [95,100]"

    */
}
```

Check container element and print report

```
// define compound validator of container elements
auto v=validator(
                _["field1"](gte,"xxxxxx")
                 ^OR^
                _["field1"](size(gte,100) ^OR^ value(gte,"zzzzzzzzzzzz"))
            );

// apply validator to container and construct validation error message

error_report err;
std::map<std::string,std::string> test_map={{"field1","value1"}};
validate(test_map,v,err);
if (err)
{
    std::cerr << err.message() << std::endl;
    /* prints:

    "field1 must be greater than or equal to xxxxxx OR size of field1 must be greater than or equal to 100
OR field1 must be greater than or equal to zzzzzzzzzzzz"

    */
}
```

Check custom object property and print report

```
// define structure with getter method
struct Foo
{
    bool red_color() const
    {
        return true;
    }
};

// define custom property
DRACOSHA_VALIDATOR_PROPERTY_FLAG(red_color,"Must be red","Must be not red");

// define validator of custom property
auto v=validator(
    _[red_color](flag,false)
);
```

```
// apply validator to object with custom property and construct validation error message

error_report err;
Foo foo_instance;
validate(foo_instance,v,err);
if (err)
{
   std::cerr << err.message() << std::endl;
   /* prints:

   "Must be not red"

   */
}
```

Pre-validate data before updating object's member

```
// define structure with member variables and member setter method
struct Foo
{
   std::string bar_value;

   uint32_t other_value;
   size_t some_size;

   void set_bar_value(std::string val)
   {
      bar_value=std::move(val);
   }
};

// define custom properties
DRACOSHA_VALIDATOR_PROPERTY(bar_value);
DRACOSHA_VALIDATOR_PROPERTY(other_value);

// template specialization for setting bar_value member of Foo
DRACOSHA_VALIDATOR_NAMESPACE_BEGIN

template <>
struct set_member_t<Foo,DRACOSHA_VALIDATOR_PROPERTY_TYPE(bar_value)>
{
   template <typename ObjectT, typename MemberT, typename ValueT>
   void operator() (
         ObjectT& obj,
         MemberT&&,
         ValueT&& val
      ) const
   {
      obj.set_bar_value(std::forward<ValueT>(val));
```

```
    }
};

DRACOSHA_VALIDATOR_NAMESPACE_END

using namespace DRACOSHA_VALIDATOR_NAMESPACE;

// define validator of custom properties
auto v=validator(
    _[bar_value](ilex_ne,"UNKNOWN"), // case insensitive lexicographical not equal
    _[other_value](gte,1000)
);

Foo foo_instance;

error_report err;

// call setter with valid data
set_validated(foo_instance,bar_value,"Hello world",v,err);
if (!err)
{
    // object's member is set
}

// call setter with invalid data
set_validated(foo_instance,bar_value,"unknown",v,err);
if (err)
{
    // object's member is not set
    std::cerr << err.message() << std::endl;
    /* prints:

     "bar_value must be not equal to UNKNOWN"

     */
}
```